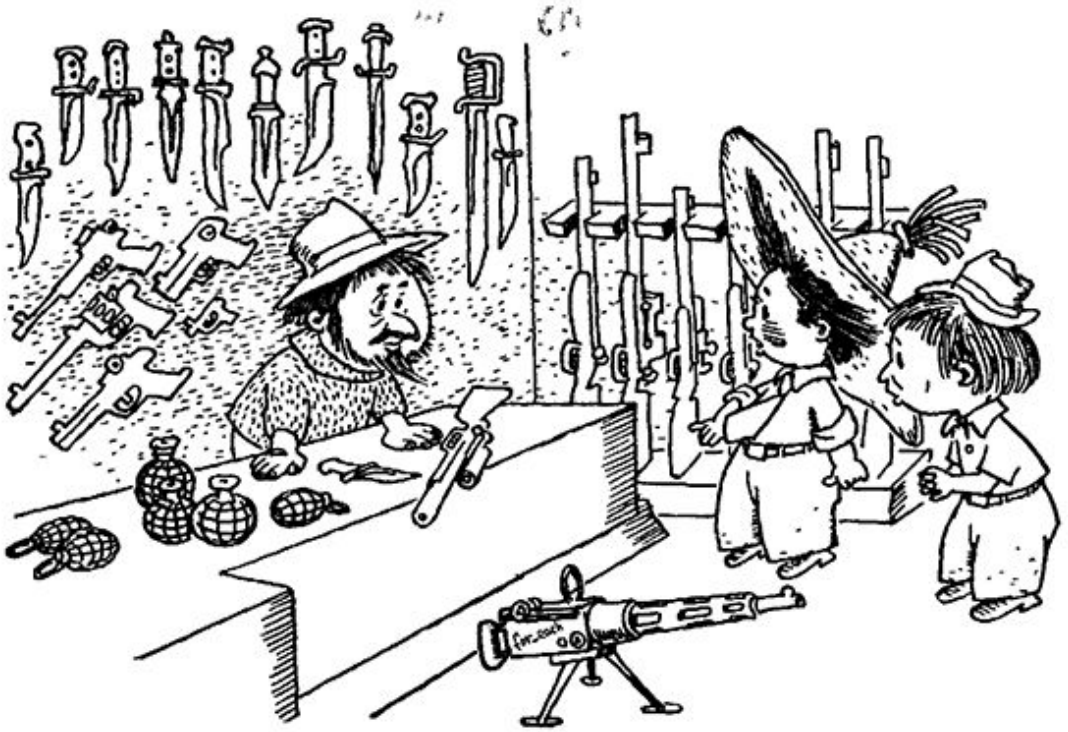

ДоБроЕ УтРо!
_paRaNoid DucK production _ 07.07.2004

STL

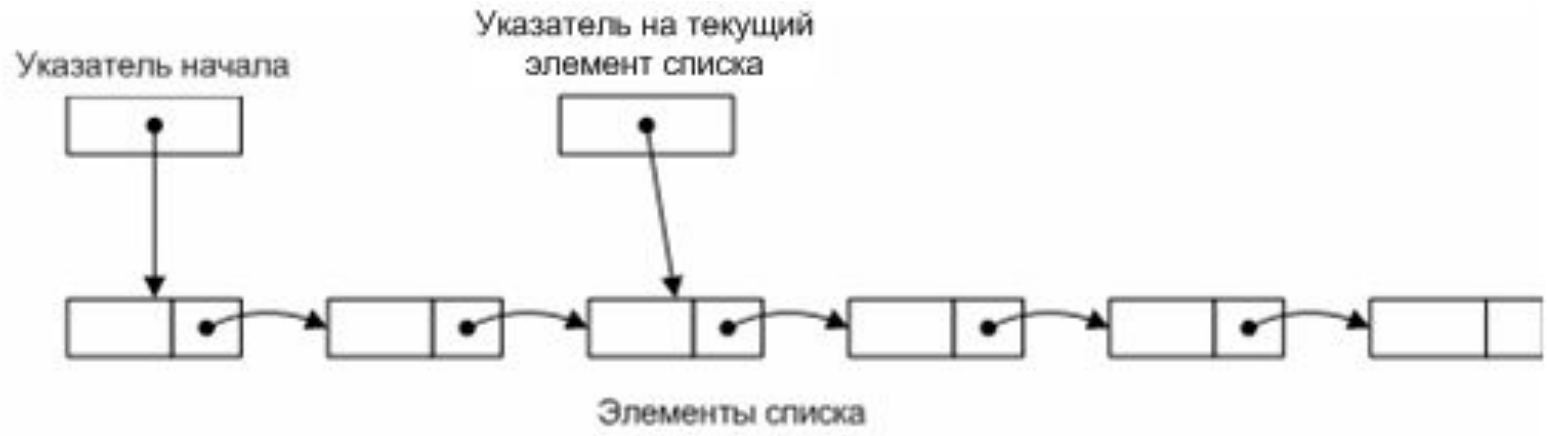


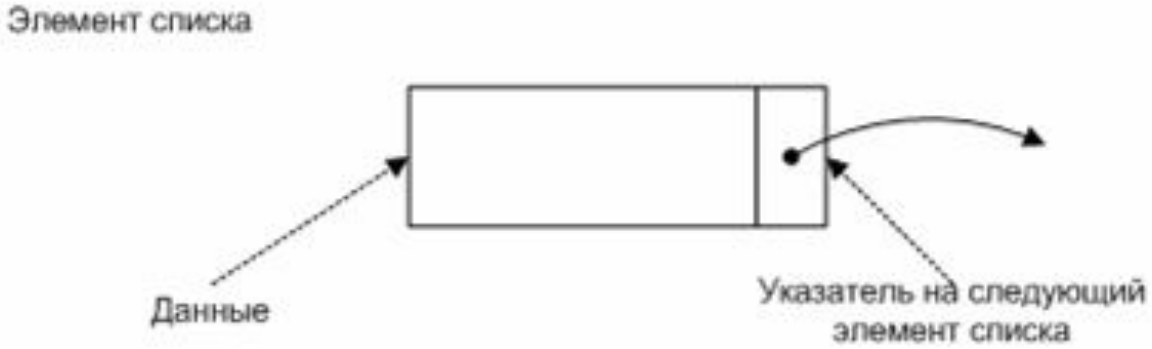
Что входит в STL

- Контейнеры
- Итераторы
- Алгоритмы
- Умные указатели, функторы...

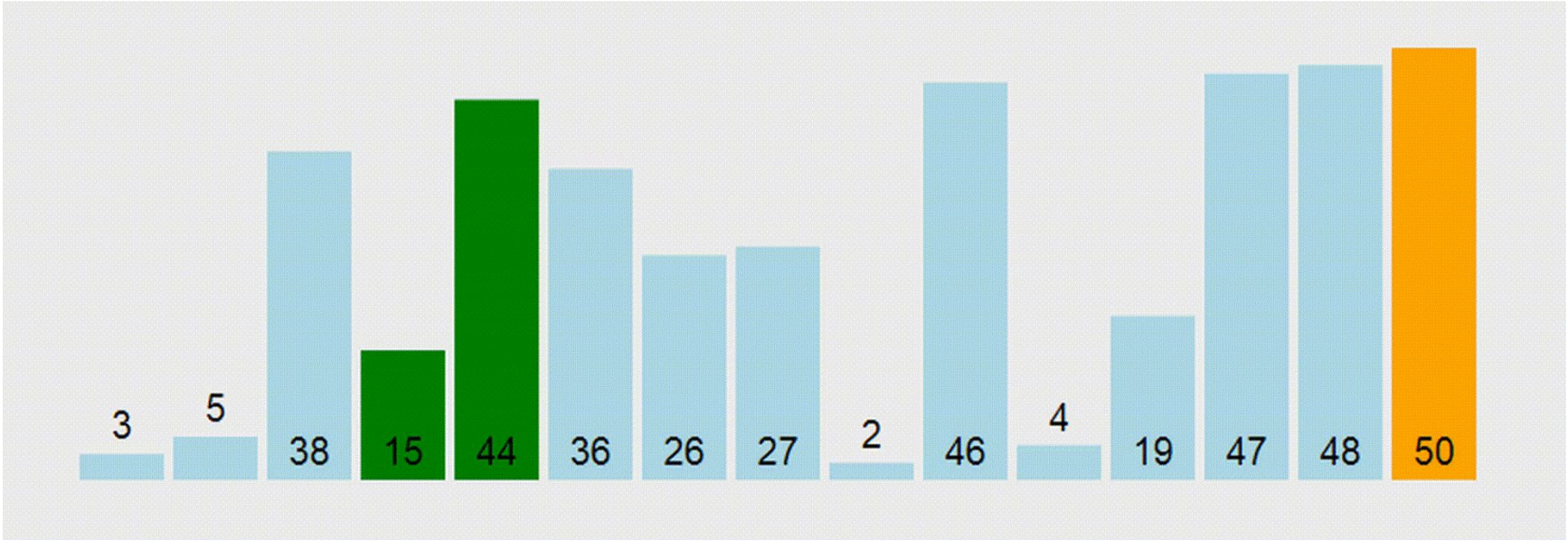
Контейнер - массив

Контейнер - список





Шаблоны алгоритмов



Итераторы

- Указывают на элементы контейнеров
- Обеспечивают доступ к элементам
- Могут перемещаться от элемента к элементу
- Их можно сравнивать между собой
- У каждого контейнера свой тип итераторов

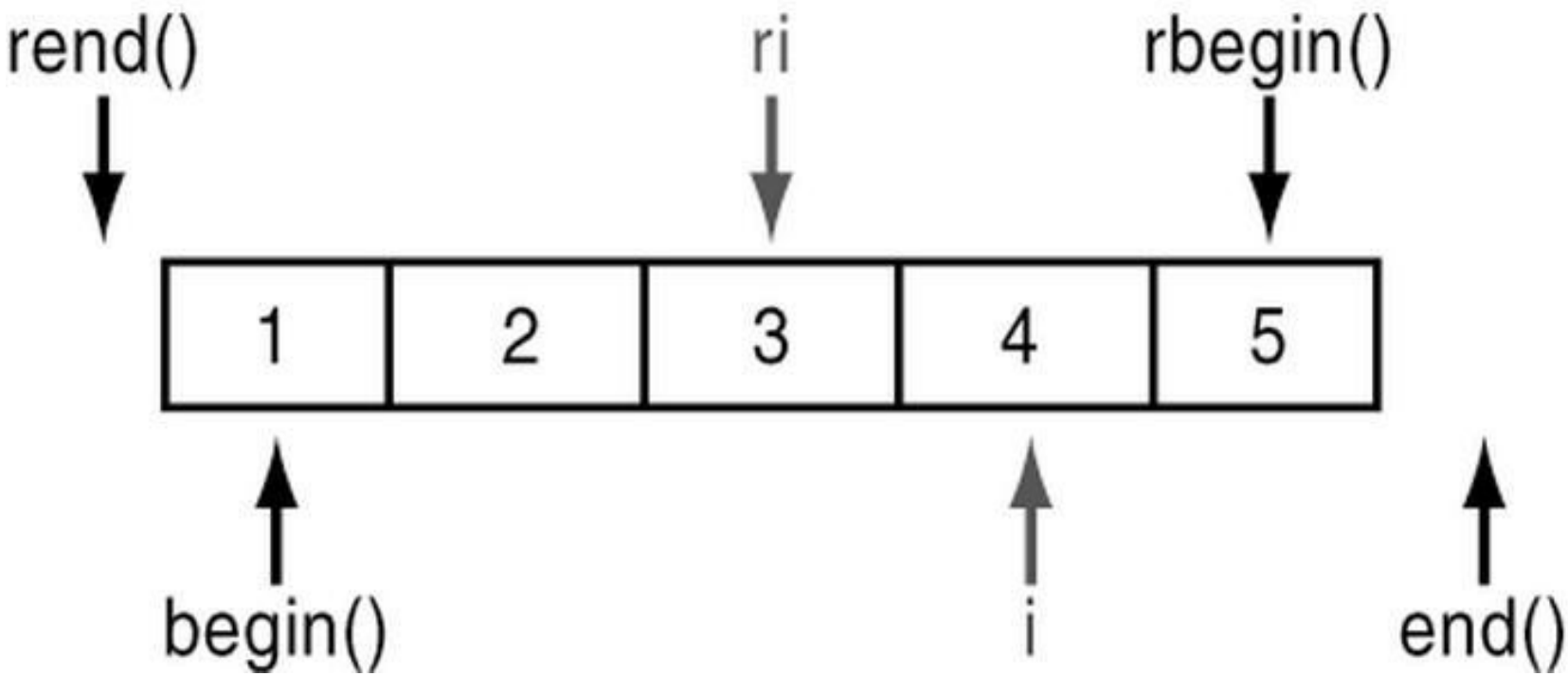
Итераторы бывают

- Последовательные / произвольного доступа
- Однонаправленные / двунаправленные
- Прямые / обратные
- Модифицирующие / только для чтения
- Особые итераторы для вставки элементов

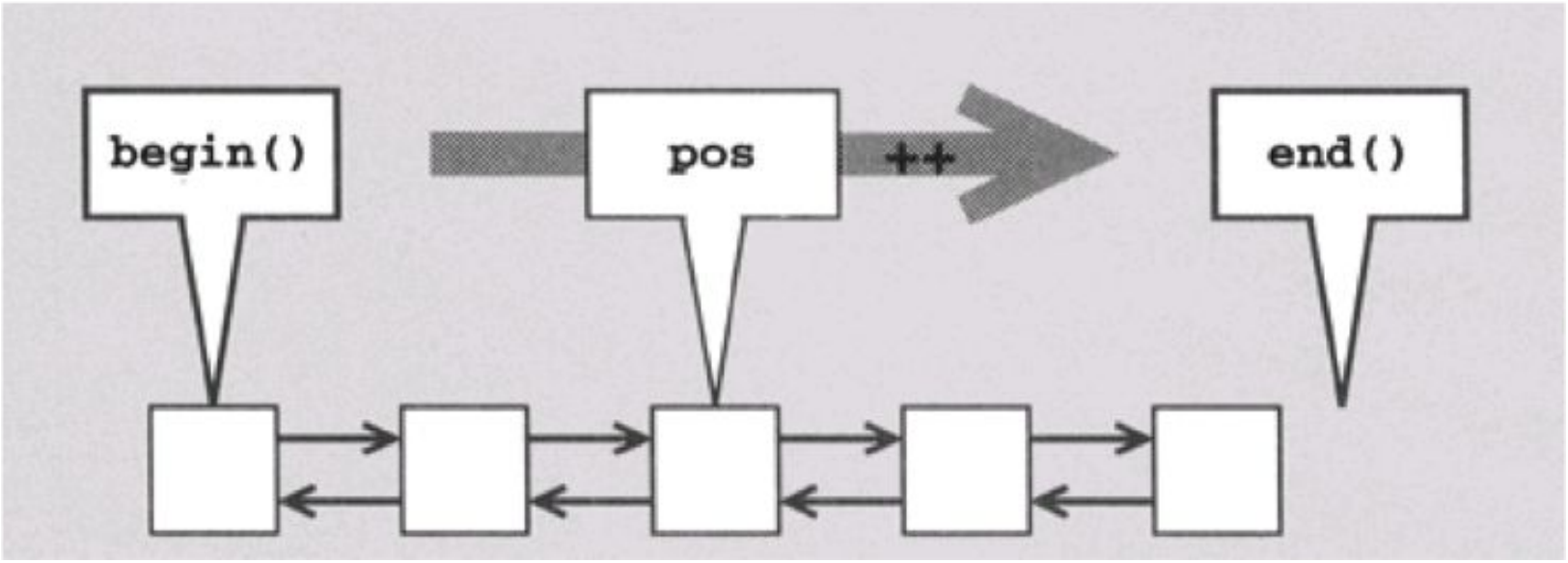
Синтаксис как у указателя

```
Iterator it1, it2;
it1->field; value = *it1; *it1 = value;
++it1; it1++; --it1; it1--; it1 += 3;
it1==it2; it1!=it2; it1>it2; it1<it2;
```

Границы контейнеров



Границы контейнеров



Диапазон значений

$$[\, a, b \,)$$

Контейнеры бывают

- Последовательные контейнеры:
  vector, array, deque, list, forward_list
- Ассоциативные контейнеры:
  set, multiset, map, multimap,
  unordered_set, unordered_map
- Контейнеры - адаптеры:
  stack, queue, priority_queue

У контейнера есть

- Типы данных:
  iterator, const_iterator, reverse_iterator
- Методы, возвращают итераторы:
  begin(), rbegin(), cbegin()
  end(), rend(), cend()
- Методы, возвращают размер:
  size(), empty(), capacity()

У контейнера есть

- Управление размером:
  clear (), resize(), reserve()
- Вставка:
  push_back(), push_front(), insert()
- Удаление:
  pop_back(), pop_front(), erase()

```
#include <iostream>
#include <vector>
//using namespace std;
int main ( )
{
    std::vector<int> v = {7, 5, 16, 8};
    v.push_back(25);
    v.push_back(13);
    std::vector<int> :: iterator it;
    for ( it = v.begin(); it != v.end(); ++it )
    {
     std::cout << *it << '\n';
    }
}
```

```
#include <iostream>
#include <vector>
//using namespace std;
int main ( )
{
    std::vector<int> v = {7, 5, 16, 8};
    v.push_back(25);
    v.push_back(13);
     for ( int n : v )
    {
      std::cout << n << '\n';
    }
}
```

```
#include <iostream>
#include <vector>
//using namespace std;
int main ( )
{
    std::vector<double> v;
    v.resize(10);
    for ( int i = 0; i < v.size(); ++i )
    {
     v[i] = 1.0 / i;
    }
}
```

```
#include <vector>
#include <iostream>

class Info
{
    std::string   m_FIO;
    double        m_Mark;
public:
     Info() : m_FIO(), m_Mark(0)
      {  std::cout << "Call: Info()" << std::endl; }
     Info(const char* fio, double mark) : m_FIO(fio), m_Mark(mark)
      {  std::cout << "Call: Info(fio,mark)" << std::endl; }
     Info(const Info& obj ) : m_FIO(obj.m_FIO), m_Mark(obj.m_Mark)
      {  std::cout << "Call: Info(Info&)" << std::endl; }
};
```

Контейнер объектов

```
    std::vector<Info> data;
//1)
    Info   object( "Иванов", 3.5 );  // Call: Info(fio,mark)
    data.push_back( object );      // Call: Info(Info&)
//2)
    data.emplace_back( "Петров", 4.3 ); // Call: Info(fio,mark)

//3)
    data.reserve(10);
    data.resize   (2);    // Call: Info(), Call: Info(Info&), Call: Info(Info&), Call: Info(Info&)
```

Контейнер указателей

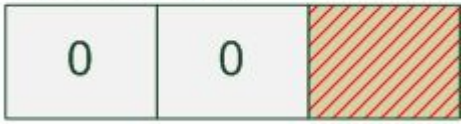
```
    std::vector<Info*>  data;
//1)
    data.push_back( new Info( "Иванов", 3.5 ) );  //Call: Info(fio,mark)
//2)
    data.emplace_back( new Info( "Петров", 4.5 ) ); // Call: Info(fio,mark)

//3)
    data.reserve(10);
    data.resize   (2);
```

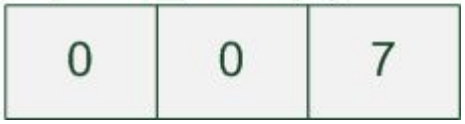
Аллокация и реаллокация

std::vector<int> v;
v.reserve(3);

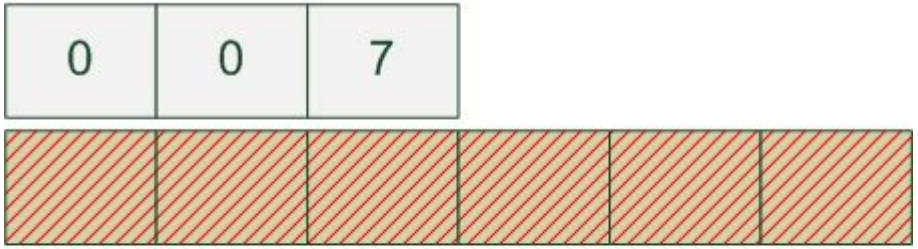
v.resize(2);

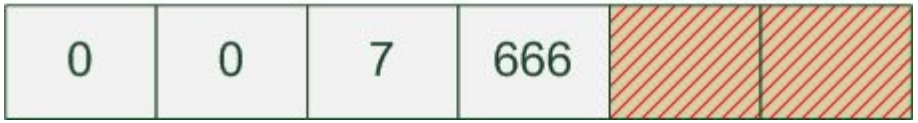


v.push_back(7);



v.push_back(666);





функторы

- Функция (по имени)
- Объект класса, с перегруженным оператором вызова функции ()
- Лямбда выражение

функтор - функции

std::vector< std::string > text;

```
bool compareByLength ( const std::string& left,  const std::string& right)
{
    return left.length() < rigth.length();
}
std::sort( text.begin(), text.end(), compareByLength );
```

```
std::sort( text.begin(), text.end(),
      [](const std::string& left,  const std::string& right)
      {
          return left.length() < rigth.length();
      });
```

функтор - объект класса

```
struct StringCompareByLength
{
    bool operator() ( const std::string& left,
                      const std::string& right ) const
    {
      return left.length() < rigth.length();
    }
};
StringCompareByLength  compare;
if( compare( "Котик", "Собачка" ) )
{
    //мы сюда попадем, котик меньше собачки по длине слова
}

std::sort( text.begin(), text.end(), compare);
```

Шаблон функтора Less

```
template <class T>
struct less
{
    bool operator() ( const T& left, const T& right ) const
    {
      return left < rigth;
    }
};

std::less<std::string> compare;
if( compare( "Котик", "Собачка" ) )
{
    //снова сюда попадем, котик раньше собачки по алфавиту
}
std::sort( text.begin(), text.end() );
```

std::set std::multiset

```
template<
        class Key,
        class Compare = std::less<Key>,
        class Allocator = std::allocator<Key>
> class set;
```

```
#include <set>
std::set<int>   cont1;
std::set< std::string, StringCompareByLength>  cont2;
```

класс - функтор

```
struct StringCompareByLength
{
    bool operator() ( const std::string& left,
                      const std::string& right ) const
    {
      return left.length() < rigth.length();
    }
};

StringCompareByLength compare;
std::sort( text.begin(), text.end(), compare);
std::sort( text.begin(), text.end(), StringCompareByLength() );

std::set< std::string, StringCompareByLength>  cont2;
```

std::set std::multiset

- **int count (const Key&)**
- **iterator find (const Key&)**
- std::pair<**iterator**, **iterator**> **equal_range (const Key&)**
- **iterator lower_bound (const Key&)**
- **iterator upper_bound (const Key&)**

std::map std::multimap

- template<
  class Key, class T,
  class Compare = std::less<Key>,
  class Allocator = std::allocator<std::pair<const Key, T> >
  > class map;

```
#include <map>
std::map<std::string, int>   cont1;
std::map<int, std::string>   cont2;
```

std::map std::multimap

```
template< class T1, class T2 >
struct pair
{
        T1 first;
        T2 second;
};

template< class T1, class T2 >
std::pair<T1,T2> std::make_pair( T1 first, T2 second );
```

std::map std::multimap

- **int count (const Key&)**
- **iterator find (const Key&)**
- std::pair<**iterator**, **iterator**> **equal_range (const Key&)**
- **iterator lower_bound (const Key&)**
- **iterator upper_bound (const Key&)**
- **Value operator[] (const Key& key)**

```
#include <map>
typedef  std::map<std::string, int>     ContainerType;
ContainerType                   info;
ContainerType::iterator         it;
```

```
it = info.find( str );
if( it == info.end() )
    info.insert( std::make_pair( str, 1) );
else
    it->second++;
```

++ info[str];